Elastic Application-Level Monitoring for Large Software Landscapes in the Cloud

Florian Fittkau, Wilhelm Hasselbring

Elastic Application-Level Monitoring for Large Software Landscapes in the Cloud

Florian Fittkau and Wilhelm Hasselbring

Software Engineering Group, Kiel University
24098 Kiel, Germany
{ffi, wha}@informatik.uni-kiel.de

Abstract. Application-level monitoring provides valuable, detailed insights into running applications. However, many approaches often only employ a single analysis application. This analysis application may become a performance bottleneck when monitoring several programs resulting in reduced monitoring quality or violated service level agreements of the monitored applications.
We present an approach for elastic, distributed application-level monitoring for large software landscapes consisting of several hundreds of applications by utilizing cloud computing. Our approach dynamically inserts and removes worker levels to circumvent overloading the analysis master application without interrupting or pausing the actual live analysis of the monitored data. To evaluate our approach, we conduct an experiment in which we generate load – following a real workload pattern – on web applications in a 24 hour experiment.
In our experiment, 160 monitored JPetStore instances generate roughly 20 million analyzed method calls per second in the peak. Furthermore, two worker levels are dynamically started and removed in line with the imposed workload on the monitored applications. The experiment shows that our monitoring approach is capable of live analyzing several millions of monitored method calls per second without overloading the analysis master application.

Keywords: Application-Level Monitoring, Elasticity, Cloud Computing

1 Introduction

Enterprises often run and administer large and complex software landscapes featuring hundreds of running applications [12]. Since most of them evolve over decades, the comprehension of those landscapes often gets lost due to missing documentation, changing business requirements, or employees, for example. Application-level monitoring can support in the comprehension process such large software landscapes [6]. However, most approaches only feature one analysis node for the monitored data which provides poor scalability – especially in cloud environments where the monitored applications adapt to the imposed workload.

In this paper, we present an elastic, distributed application-level monitoring approach to circumvent this overuse of a single analysis node. Our approach dynamically inserts and removes preprocessing worker levels depending on the actual utilization of the analysis master. The change of the system takes place without interrupting the actual analysis of the monitored data.

Furthermore, we present a thorough evaluation of the described approach in which our monitoring solution monitors 160 elastically scaled web applications and analyzes several millions of method calls per second. To facilitate the verifiability and reproducibility of our results, we provide a data package [4] containing all our experimental results and source code.
In summary, our main contributions are:

- an elastic, distributed application-level monitoring approach which dynamically inserts and removes worker levels, and
- a thorough evaluation of the approach incorporating 160 scaled web applications and several millions of analyzed method calls per second.

The remainder of this paper is organized as follows. Section 2 states the addressed problem. Afterwards, our approach for elastic, distributed application-level monitoring is described. A 24 hour experiment for applying the concept is presented in Section 4 as evaluation. Related work is discussed in Section 5. Finally, we draw the conclusions and illustrate future work in Section 6.

2 Problem Statement

Employing only a single analysis node for live processing the monitoring data can easily become a bottleneck. For example, in our evaluation described in Section 4, the analysis would operate at full capacity after receiving load from only four monitored applications. In general, this number is determined by the amount of monitoring and the hardware for the analysis node. However, eventually every node will be fully utilized if the workload rises to some point. Cloud computing aims to provide – perceived – infinite scalability for the monitored applications and therefore, this should also apply to the monitoring solution.

Application-level monitoring tools, e.g., Kieker [9], typically offer three configurable strategies, what should be done when the analysis cannot process the current monitoring data. The first strategy simply terminates the monitoring. Since this requires a manual restart of the application to start monitoring again, this behavior is undesirable for a high monitoring quality. However, this typically does not affect the service-level agreements (SLAs) of the monitored applications.

The second strategy discards new monitoring records until a space in the monitoring queue is available. Therefore, this behavior is similar to sampling which only monitors method calls on a defined interval, e.g., every 10th request. This strategy typically imposes no SLA violations at the expense of a reduced monitoring quality. However, it can automatically recover when the workload drops and thus is typically preferable over the first strategy and therefore, often employed in practice.

The third strategy uses blocking until a free space in the monitoring queue becomes available. While this behavior seems appealing on first sight, it can violate the SLAs when the analysis node takes a long time to recover from its high workload. The SLA violations are caused by the waiting of the application for finishing the writing of the monitored data. Therefore, it is not processing user requests often leading to loss in revenue due to annoyed customers.

This situation can become even more expensive, if the capacity manager utilizes the waiting user requests for its upscaling condition for the applications. Since only one analysis node is employed, the newly started application would also wait for the analysis node to finish. Therefore, the capacity manager might keep starting new instances until some node limit is reached and the service provider has to pay for application nodes that are waiting for the analysis of the monitoring data.

Based on the chosen strategy, either the quality of the SLAs or the monitoring quality is reduced in the circumstance of a fully utilized analysis node. One way to postpone this problem is an analysis node with a high number of CPU cores and a high amount of RAM. However, the analysis must be designed to utilize an infinite number of CPU cores and if the workload rises, the number of CPU cores must be increased according to the peeks in the workload. Hence, they become superfluous during low workload.

3 Elastic, Distributed Application-Level Monitoring

In this section, we describe how we employ an elastic, scalable monitoring approach to circumvent the overutilization of a single analysis node by dynamically adding or removing preproccesing levels.

We start by outlining our basic idea. Then, our general scalable architecture is described. Afterwards, the analysis component, which enables the connection of multiple analysis workers in series, is explained. Then, we illustrate the scaling process for multiple worker levels. Lastly, assumptions and limitations of our approach are discussed.

3.1 Idea

Fig. 1 illustrates the basic idea of our elastic, distributed application-level monitoring. When the analysis master impends to become overutilized, a new worker level is dynamically added in front of it. Similar to the MapReduce pattern [3], each worker on the new level analyzes one part of the monitoring data. To circumvent an overutilization of the workers, the associated worker applications are scaled within their worker level. With this preprocessing step, the `Master` is only required to combine the analysis results. Eventually with rising workload, the merging of the results impends to overload the `Master` again. Then, a second level is dynamically inserted between the first level and the `Master`. In theory, this behavior can continue infinitely.

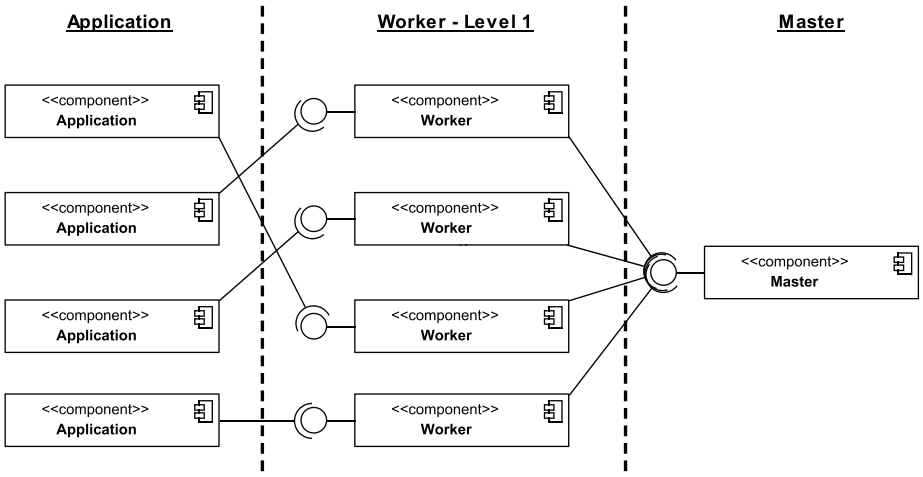

Fig. 1. Basic idea of dynamic worker levels

3.2 Scalable Architecture

Next, we show our general scalable architecture. Fig. 2 displays this architecture including our capacity manager `CapMan` and one master node. Therefore, it represents the initial state when only a small amount of monitoring data has to be analyzed. In our architecture, the capacity manager includes the workload generation and its load balancing for our experiment due to convenience reasons. Therefore, the applications are accessed by `CapMan` to simulate user requests. A `System Monitor` records the CPU utilization of the application nodes and sends this utilization to `CapMan`. `CapMan` uses these values, in addition to the outstanding request count from the workload generation, for scaling the applications. This cycle forms the employed load generation on the applications and their automatic elastic scaling.

Every application contains a `Monitoring` component. At its start, it requests an IP address from the `Monitoring LoadBalancer`. This request contains a `loadbalancing group` property to determine the kind of application which the `Monitoring` component wants to access. For example, the applications use `analysis` to reflect their wish to write monitoring data on an analysis node. In Fig. 2, the shown state only exists of one analysis node, i.e., the master node. Therefore, the `Monitoring` component receives the IP address of the master node and sends its monitoring data to the master analysis application. After a defined interval, the `Monitoring` component again fetches an IP address from the `Monitoring LoadBalancer` and if necessary connects to the newly received IP address. Therefore, the monitoring data is distributed to different nodes when multiple nodes (e.g., on a worker level) are available. This results in an approximate equal utilization of the target nodes. Similar to the application nodes, the CPU utilization of the analysis nodes is sent by a `System Monitor` to `CapMan`

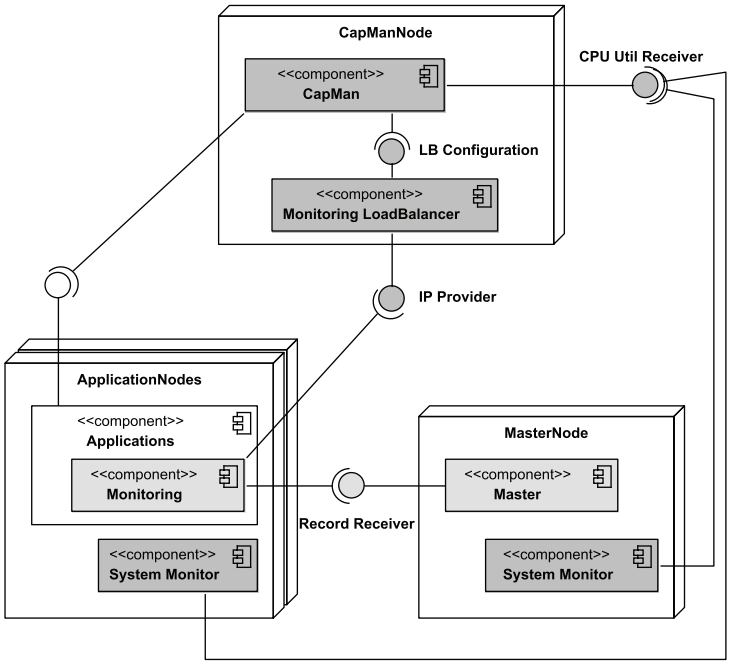

Fig. 2. Our scalable monitoring architecture

which uses these values for scaling the analysis nodes. If a new analysis node is started by `CapMan`, the IP address of the newly started node is registered in the `Monitoring LoadBalancer` under a defined `loadbalancing group` property.

3.3 Analysis Component

To enable a series connection of the different worker levels, the analysis component follows the activities shown in Fig. 3. The monitoring data is received via a TCP connection and a record reconstruction step creates record objects. A record object can contain, for example, a monitored method call (and its data like the method duration), an ID-to-String-mapping, or general meta data. ID-to-String-mappings are an important concept for reducing the transferred data by replacing Strings with an integer representation before sending. The record objects are passed to the trace reconstruction step which links the loose method call records to an execution trace representing the full execution path of one user request. Afterwards, the traces are passed to a trace reduction activity. The chance of same traces typically increases when multiple user requests are conducted. For example, most of the users will access the main page of a website which will often generate the same execution trace in an application. To save network bandwidth and CPU cycles on the next analysis node in the chain, similar traces are reduced to one trace class. For monitoring how many times the

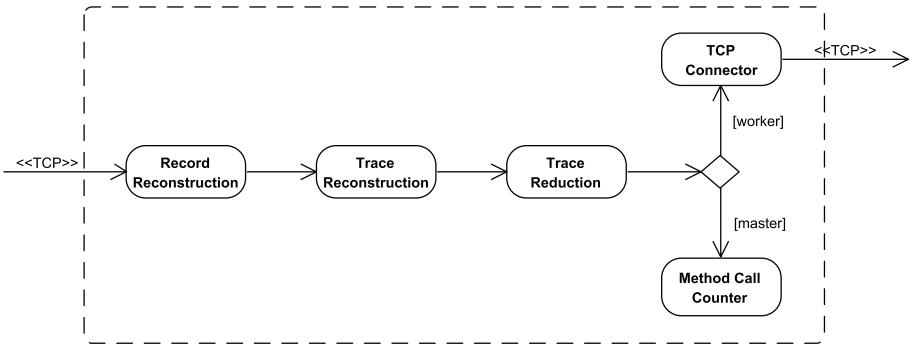

Fig. 3. Activities in the analysis component (worker and master)

trace class was called, it contains an attribute *called times* and runtime statistics (e.g., minimum and maximum duration) for the monitored method calls. To be able to determine which host might behave differently, the runtime statistics are formed on a per host basis.

If an analysis node is started as a worker node, the trace classes are sent to the next analysis node in line via a TCP connector which sends these trace classes as serialized single record objects again. If the analysis node is running as the master node, it simply counts the processed monitored method calls in our example. However, these trace classes can also be used for, e.g., creating a model of the monitored applications as we do it for our ExplorViz [6] visualization.

3.4 Scaling Process

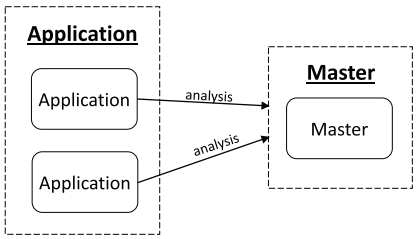

Fig. 4. Initial state before scaling

In Fig. 4, the state from Fig. 2 is visualized in a simplified form. The boxes with dashed lines represent one scaling group, i.e., a group of applications which is scaled independently by a capacity manager. The name of each scaling group is displayed at the top. There are two scaling groups: Application and Master. Arrows illustrate accesses to the target scaling group. The label of an arrow is the

`loadbalancing group` name used to request an IP address from the `Monitoring LoadBalancer`. In the initial state, the applications access the `Master` scaling group by using the loadbalancing group name *analysis*. Decoupling the scaling group name and the loadbalancing group name enables the worker levels to get dynamically inserted or removed between each processing level.

Upscaling Fig. 5 illustrates the process of dynamically adding one worker level. After the CPU utilization of the `Master` rises over a defined threshold, this process is triggered. At first, a new loadbalancing group is created which is named *worker-1* and contains the `Master`. Then, *two* new worker nodes are started. We assume the same configuration on each analysis node. Therefore, starting only one worker node would result in the same high CPU utilization encountered on the `Master`. The new worker nodes send their data to the scaling group which is resolved by the loadbalancing group name *worker-1*. This state is visualized in Fig. 5a. After the worker application on the nodes are started, the two workers are added to the loadbalancing group *analysis* and the `Master` is removed from it. The final state is illustrated in Fig. 5b. Notably, the order of adding and removing loadbalancing groups is important because the analysis should not be paused during the scaling process.

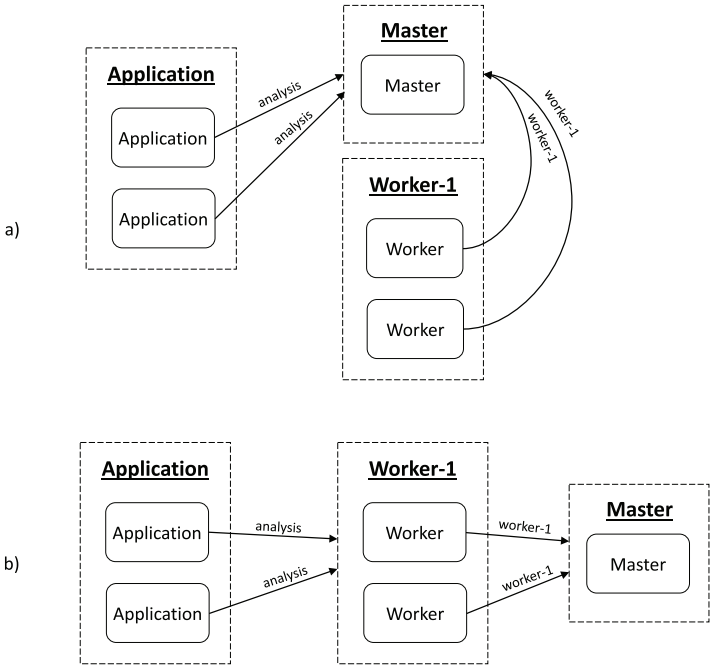

Fig. 5. Activities forming the upscaling process

Downscaling The downscaling process follows the upscaling process in reverse order. However, we employ a different scaling condition. Our first approach was using the analogous CPU utilization of the `Master` when it falls below a defined threshold. However, this condition is independent from the amount of nodes in the previous worker level. Therefore, it would also trigger when the previous worker level contains, e.g., 10 nodes and shutting down all of them would typically result in an overutilization of the `Master`. This could be lifted by only downscaling when there are exactly two nodes of the previous worker level left. However, this still contains no statement about the utilization of the previous worker level. For example, both workers might be heavily utilized. Hence, we use the CPU utilization of the previous worker level as downscaling condition. When only two nodes are left in the previous worker level and the average CPU utilization falls below a defined threshold in this scaling group, it is shut down and removed by following the upscaling process in analogous reverse order. Therefore, downscaling is not delaying or pausing the analysis either.

3.5 Assumptions and Limitations

For being able to reduce the traces on one worker, similar execution traces have to be generated by the applications in one processing interval (e.g., 5 seconds). From our observations, web applications often impose similar traces if the behavior is not user-specific. However, if every trace is different from another, our worker concept will not work.

A further limitation is imposed by the round-robin connection of the worker nodes. Each worker connects to a new node on a regular basis. Therefore, a common state between the worker and its target node has to be reestablished each time. For instance, the ID-to-String-mapping is shared between both nodes. Therefore, for each new connection this mapping must be communicated to the target node. If this exchange takes too much processing time, our approach may not work. This limitation could probably be lifted by proper caching techniques which stays as future work.

4 Experimental Evaluation

In this section, we present an experiment for evaluating our elastic, distributed application-level monitoring approach. We start by describing the used workload curve and the experimental setup. Then, the results of the experiment are discussed and a summary is presented. At last, we identify threats to validity.

4.1 Workload

Our employed workload curve can be seen in Fig. 6. The access pattern of our object system was modeled after a real web application access pattern which is detailed in [11].

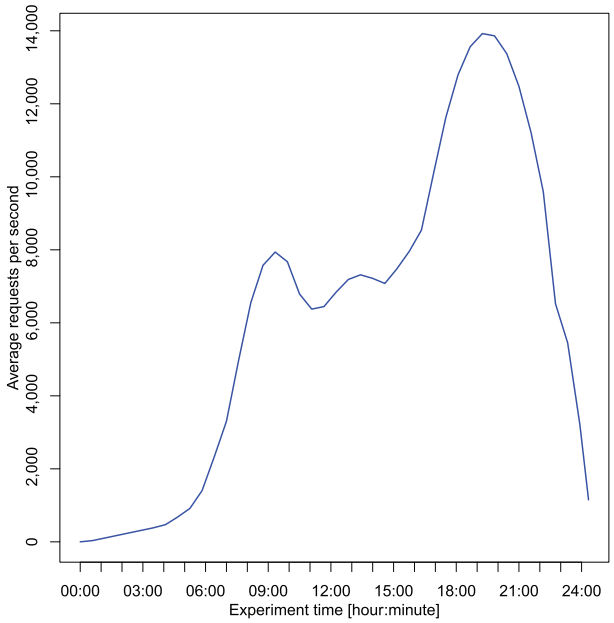

Fig. 6. Employed workload curve

The workload curve represents a day-night-cycle workload pattern which can be considered typical for regional websites. It starts with a rising workload until six o'clock when about 1,000 requests per second are conducted. Then, the load peaks at nine o'clock with about 8,000 requests per second. Afterwards, it slightly decreases to about 7,000 requests per second. In the evening at around eight o'clock p.m., the request count peaks with about 14,000 requests per seconds. Then, it falls to about 1,000 requests per second at midnight and shortly behind this point in time, it drops to no requests for our experiment.

4.2 Experimental Setup

We utilize our private cloud running OpenStack[1] containing seven servers. Each server has two Intel Xeon E5-2650 (2.8 GHz, 8 cores) CPUs, 128 GB of RAM, and a 400 GB SSD. Therefore, the total amount of resources are 112 CPU cores, 896 GB of RAM, and 2.8 TB of disc space. Since every core also features Hyper-threading, we configured our cloud to have a maximum of 224 virtual cores.

As object system, we utilize the web application JPetStore[2] written in Java. As the name suggests, it is a software for setting up a small web shop for pets. We monitor all method calls in the *com.ibatis* package which contains source code

[1] `https://www.openstack.org`
[2] `http://ibatisjpetstore.sf.net`

written by the authors of JPetStore and all method calls in the *org.apache.struts* package which significantly contributes to the generation time of one web page.

Two flavors – resource configurations in OpenStack terms – are used in our experiment. The first one is a small flavor which is used by every dynamically started instance (Master, Worker, and JPetStore nodes). It consists of one virtual CPU (VCPU), 3 GB of RAM, and 10 GB disc space. With this configuration, we are able to start a total count of 224 possible instances. The second flavor is only used by the capacity manager node. Since this node also contains the `Monitoring LoadBalancer` and generates the workload, it should be guaranteed to have sufficient resources for its tasks. Therefore, the capacity manager node runs with 8 VCPUs, 16 GB of RAM, and 80 GB disc space which reduces the maximum count of dynamically started instances of the small flavor to 216.

A large setup cost for this experiment was imposed by tuning the operating system of the physical server (Debian Sid) to process the large amount of requests per second. In the default configuration, this request amount is detected as potential denial of service attack and thus the requests are dropped. For example, we had to tune the number of usable TCP ports, TCP state timeouts, the maximum open files, and the NAT connection tracking tables. For potential replications, our experimental package contains the relevant configuration files. Furthermore, we provide the virtual machine image used for all our instances to reduce the setup costs.

The configuration of our capacity manager `CapMan` contains three scaling groups, i.e., for the Master, the workers (as prototype for dynamically started levels), and the JPetStores. The Master scaling group uses a threshold of 40 % average CPU utilization to trigger the insertion of a new worker level. `CapMan` always calculates the average CPU utilization over a time window of 120 seconds to reduce the impact of short utilization spikes. The prototype of a worker scaling group is configured with a downscaling condition of a value below 15 % average CPU utilization. A new instance is started if the average CPU utilization is above 45 %. In the JPetStore scaling group, an instance is shut down when the average CPU utilization falls below 27 %. For upscaling, the outstanding requests are counted and when these are above 200, a new instance is started. In contrast to the other scaling groups, the start time of a new instance is not negligible. Therefore, 16 seconds are waited during booting since Jetty must be started and JPetStore must be deployed.

4.3 Results and Discussion

Fig. 7 shows the resulting JPetStore instance count and the average CPU utilization of the Master node in our experiment. In general, the count of the JPetStore instances follows the workload curve and peaks at 160 instances. The only exception is the instance count not reducing after the first peak in the workload at hour nine. This is caused by the 27 % average CPU utilization downscaling condition which could be further reduced to also scale down in this situation.

Notably, since the workload curve is reflected in the JPetStore instances, the general scaling in accordance to the imposed workload is functioning. We now

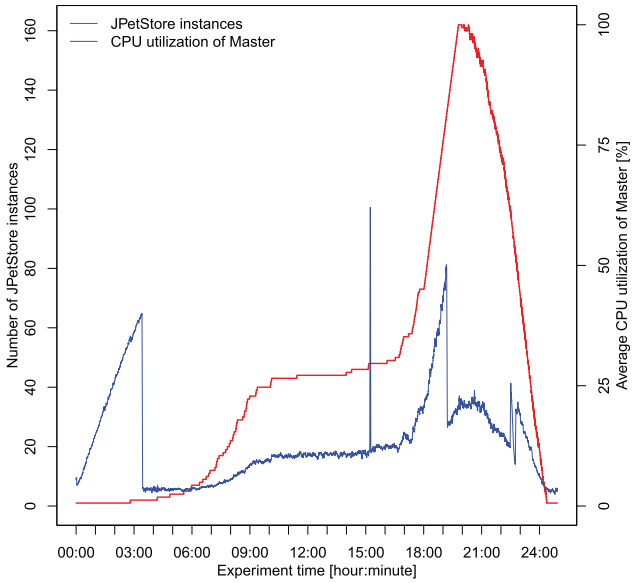

Fig. 7. JPetStore instance count and average CPU utilization of Master node

take a closer look at the CPU utilization of the Master, the started worker levels, and the monitored method calls per second.

With the constantly rising workload, the CPU utilization of the Master also constantly increases until approximately hour three. At this time, a new worker level is started since the average CPU utilization of the Master rises above 40 %. The started analysis nodes are visualized in Fig. 8 where this circumstance can also be seen. After the successful insertion of the worker level, the CPU utilization of the Master drops to about 3 %. Notably, at this point in time only two JPetStore instances are started. This is reasonable since about 400,000 method calls per seconds need to be analyzed.

After hour three, the CPU utilization of the Master node only rises slightly to 11 % while the JPetStore instance count drastically increases to about 40 instances in hour ten. The work induced by the analysis of the monitoring data is distributed to the workers in the first worker level where the instance count increases to 20 instances till hour ten.

In hour 15, a short peak of about 62 % in the Master CPU utilization can be seen. Since it only occurred for about one minute and has a difference of about 50 % to the previous and afterwards values, this peak is an anomaly. During other runs on our private cloud, we often observed this behavior when another instance is started on the same physical host. Therefore, we implemented an anomaly detection algorithm in our capacity manager for this circumstance and thus no new worker level is started in hour 15.

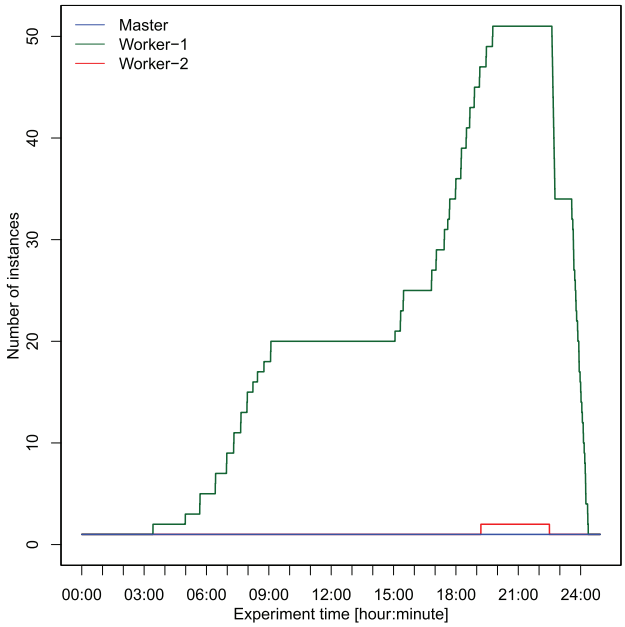

Fig. 8. Analysis nodes and number of instances in each level

The JPetStore instance count is rising again from hour 15 till hour 20 peaking in 160 instances. Therefore, the instance count of the workers in the first worker level is also increasing which peaks at about 50 instances in hour 20. Since the Master has to receive and merge the traces from those instances, its CPU utilization also rises until hour 19. Then, the CPU utilization is once again above the 40 % threshold which results in a newly inserted worker level. Afterwards, the Master CPU utilization drops to about 17 %. This drop is not as large as the previous one but still it circumvents the overutilization of the Master node.

In hour 20, the workload approximately decreases until hour 24. This leads to a reduction of the JPetStore instances and therefore, also the analysis nodes are reduced. At first, the second worker level is completely removed in hour 22 resulting in an increase of the CPU utilization on the Master node. The worker instances in the first worker level are also reduced until hour 24 is reached. Then, also the first worker level is removed resulting in the initial configuration where only the Master node is analyzing the monitored data.

Fig. 9 visualizes the monitored and analyzed method calls per second. In general, it follows the requests per second of the workload and peaks in about 20 million analyzed calls per second. The only exception is a short spike in hour 22. This resulted from a too fast shutdown of one analysis node in the second worker level which can be circumvented by increasing the shutdown delay of analysis nodes in higher worker levels.

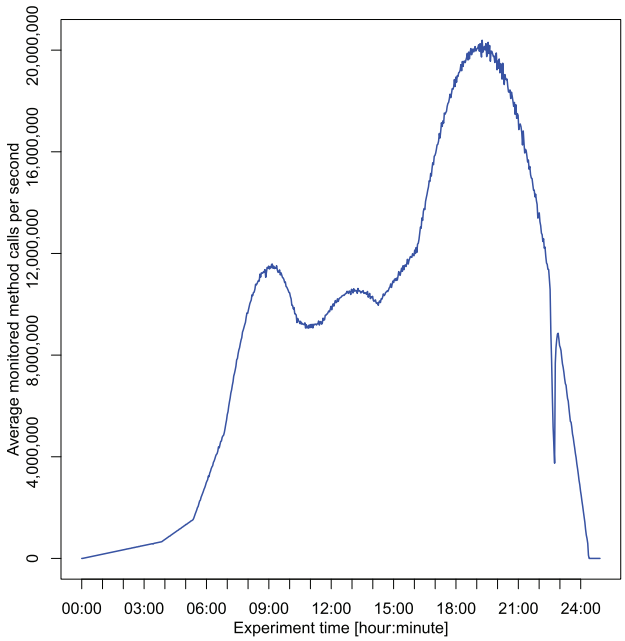

Fig. 9. Average monitored and analyzed method calls per second

4.4 Summary

Summarizing the results, our elastic, distributed application-level monitoring approach shows feasible to circumvent the overutilization of the Master node in spite of a rising workload. Furthermore, the Master node employs only a single VCPU. Therefore, during low workload on the monitored applications, the minimum costs for monitoring incur.

4.5 Threats to Validity

We conducted our experiment on our private cloud with scaling of JPetStore instances. For external validity, it should also be evaluated in other environments and with other applications. The same applies to the employed workload curve and the amount of conducted monitoring.

Our experiment involved two worker levels due to having only 216 VCPUs available. The results for a third worker level might be different. Further experiments are required to show if the third worker level still circumvents the overutilization of the Master node.

Furthermore, similar traces are generated by accessing JPetStore. We assume that our monitoring approach will behave differently if this assumption is not satisfied. This should be also investigated in further experiments.

5 Related Work

Brunst and Nagel [2] present a parallel analysis infrastructure. They focus on massive parallel systems with thousands of processor cores. In contrast, we focus on the monitoring and analysis of applications running on typical business servers or in cloud environments.

Meng et al. [10] propose a Monitoring-as-a-Service solution for monitoring cloud infrastructures. To monitor the complex infrastructure of Cloud data centers, they developed a scalable and flexible monitoring topology consisting of different services. Compared to our approach, they focus on monitoring the virtualized data center environment.

Hilbrich and Muller-Pfefferkorn [7] describe a concept of a scalable job centric monitoring infrastructure. Their approach features multiple layers of short and long time storage of the monitored data. Contrary, we directly analyze the monitored data after gathering it without a persistent storage.

The ECoWare Infrastructure [1] consists of three types of components, i.e., the execution environment, processors, and a dashboard. In contrast to our approach, they use a message bus for their analysis and do not provide multiple analysis levels.

In [5], we presented a first idea of utilizing multiple worker levels. However, at this time, the count of worker levels was statically determined at the start of the system, i.e., not adapting to the actual workload. Furthermore, we only presented the idea without evaluating the concept of worker levels.

In general, our approach exhibits similarities to the MapReduce pattern [3]. However, in contrast to our approach, it does not dynamically insert or remove preprocessing levels according to the actual workload.

Capacity management approaches utilizing the monitored data for their scaling decisions, e.g., SLAstic [8], are also related to our approach, since they must analyze the monitored data just after it was observed. To the best of our knowledge, none of these approaches utilizes dynamically inserted or removed worker levels as presented here.

6 Conclusions

In this paper, we presented our elastic, distributed application-level monitoring approach to circumvent the overuse of a single analysis master application. We dynamically insert and remove preprocessing worker levels depending on the actual utilization of the analysis master without interrupting the analysis of the monitored data. In our presented 24 hour experiment, 160 monitored JPetStore instances generated roughly 20 million analyzed method calls per second in the peak. It showed that our approach is feasible and is capable of live analyzing several millions of monitored method calls.

For replications of our experiment and extensions of our approach, we provide an experimental package [4] containing the used programs as executables, written

source code under the Apache 2.0 License, server settings, virtual machine image, and raw experimental results.

Future work includes implementing and evaluating caching techniques for enhancing the state exchange between the analysis node. Furthermore, since our experiment only involved two worker levels, we aim to conduct an even larger experiment with more worker levels to investigate the applicability to thousands of monitored applications. In addition, our approach should be evaluated with different cloud environments and different monitored applications, e.g., RUBiS, to investigate the effect of these variables.

References

1. Baresi, L., Guinea, S.: Event-based multi-level service monitoring. In: Proc. of 20th Int. Conf. on Web Services (ICWS 2013). IEEE (Jun 2013)
2. Brunst, H., Nagel, W.E.: Scalable performance analysis of parallel systems: Concepts and experiences. In: Proc. of the 10th Conf. on Parallel Computing: Software Technology, Algorithms, Architectures, and Applications. Elsevier (2003)
3. Dean, J., Ghemawat, S.: MapReduce: a flexible data processing tool. Communications of the ACM 53(1), 72–77 (Jan 2010)
4. Fittkau, F., Hasselbring, W.: Data for: Elastic application-level monitoring for large software landscapes in the cloud (2015), doi: 10.5281/zenodo.19296
5. Fittkau, F., Waller, J., Brauer, P.C., Hasselbring, W.: Scalable and live trace processing with Kieker utilizing cloud computing. In: Proc. of the Symposium on Software Performance: Joint Kieker/Palladio Days 2013. vol. 1083. CEUR Workshop Proceedings (Nov 2013)
6. Fittkau, F., Waller, J., Wulf, C., Hasselbring, W.: Live trace visualization for comprehending large software landscapes: The ExplorViz approach. In: Proc. of the 1st Int. Working Conf. on Software Visualization (VISSOFT 2013) (Sep 2013)
7. Hilbrich, M., Muller-Pfefferkorn, R.: Identifying limits of scalability in distributed, heterogeneous, layer based monitoring concepts like SLAte. Comp. Sc. 13(3) (2012)
8. van Hoorn, A., Rohr, M., Gul, I.A., Hasselbring, W.: An adaptation framework enabling resource-efficient operation of software systems. In: Proc. of the Warm Up Workshop (WUP 2009) for ICSE 2010. ACM (Apr 2009)
9. van Hoorn, A., Waller, J., Hasselbring, W.: Kieker: A framework for application performance monitoring and dynamic software analysis. In: Proc. of the 3rd Int. Conf. on Performance Engineering (ICPE 2012). ACM (Apr 2012)
10. Meng, S., Liu, L., Soundararajan, V.: Tide: Achieving self-scaling in virtualized datacenter management middleware. In: Proc. of the 11th Int. Middleware Conf. (Middleware 2010). ACM (2010)
11. Rohr, M., van Hoorn, A., Hasselbring, W., Lübcke, M., Alekseev, S.: Workload-intensity-sensitive timing behavior analysis for distributed multi-user software systems. In: Proc. of the First Joint WOSP/SIPEW Int. Conf. on Performance Engineering (WOSP/SIPEW 2010). pp. 87–92. ACM (2010)
12. Vierhauser, M., Rabiser, R., Grünbacher, P.: A case study on testing, commissioning, and operation of very-large-scale software systems. In: Proc. of the 36th Int. Conf. on Software Engineering (ICSE Companion 2014). ACM (2014)